Selective Recovery From Failures In A Task Parallel Programming Model

James Dinan, Arjun Singri, P. Sadayappan
Dept. Computer Science and Engineering
The Ohio State University
dinan,singri,saday@cse.ohio-state.edu

Sriram Krishnamoorthy
Comp. Sciences and Mathematics Division
Pacific Northwest National Laboratory
sriram@pnl.gov

***Abstract*—We present a fault tolerant task pool execution environment that is capable of performing fine-grain selective restart using a lightweight, distributed task completion tracking mechanism. Compared with conventional checkpoint/restart techniques, this system offers a recovery penalty that is proportional to the degree of failure rather than the system size. We evaluate this system using the Self Consistent Field (SCF) kernel which forms an important component in *ab initio* methods for computational chemistry. Experimental results indicate that fault tolerant task pools are robust in the presence of an arbitrary number of failures and that they offer low overhead in the absence of faults.**

***Keywords*-Parallel processing, fault tolerance, task parallelism, Global Arrays, PGAS, selective recovery**

I. Introduction

Leadership class applications continue to pursue more sophisticated scientific and engineering simulations, demanding higher performance. This performance is often achieved through larger, more complex computer systems. Unfortunately, the Mean Time Between Failures (MTBF) for such systems decreases as systems grow in size and complexity. Thus, especially for long-running computations, job failures due to hardware faults are becoming a more frequent problem.

Conventional methods for handling faults involve writing periodic checkpoints to disk either at the application level or at the operating system process level. Upon failure, the computation can be resumed from the most recent checkpoint. Unfortunately, the overhead of taking checkpoints can be large due to the high volume of I/O generated and it grows with problem and system size.

Task parallelism is a popular parallel programming model that relaxes the process centric model for parallel computing by expressing the computation as a set of tasks that can be executed concurrently. This technique has been successfully used in many applications to perform automatic scheduling and load balancing of the workload to improve performance [1].

The layer of virtualization introduced by task parallelism between the units of work and the underlying hardware resources also offers unique opportunities for fault tolerance. In this work we extend Scioto [2], a scalable task parallel programming model that interoperates with the Global Arrays (GA) [3] Partitioned Global Address Space (PGAS) toolkit. Our system performs dynamic, fine-grain tracking of task execution progress and uses this information to enable selective restart when recovering from failures that occur during task parallel regions of a program. This technique improves on conventional checkpoint/restart techniques by incurring a performance penalty for recovery that is proportional to the degree of failure rather than the full system size. In addition, the system can survive an arbitrary number of failures and the cost for maintaining recovery information is low and in-memory.

This work was supported in part by the U.S. Department of Energy (DOE) through grants DE-FC02-06ER25755 and DE-FG02-08ER25850 and by the Ohio Supercomputer Center through an allocation of computing time.

In Section II we provide an overview of the task parallel programming model, failure model and fault tolerance framework. In Section III we present a metadata scheme for fine-grain task completion tracking. In Section IV we present the fault tolerant task execution algorithm and in Section VI we evaluate our system using the Self Consistent Field kernel from the computational chemistry domain.

II. Overview

We extend the Scioto task parallel programming model [2] with fault tolerance. In this initial exploration, we place several restriction on the full task parallel programming model to better focus on core fault tolerance issues. This restricted programming model is sufficient to support a number of important computational chemistry applications and will be relaxed in future work.

A. Global Arrays

Scioto tasks operate on data stored in the global address space provided by Global Arrays (GA) [3]. GA is a Partitioned Global Address Space (PGAS) parallel programming model that provides support for distributed shared multidimensional arrays on distributed memory systems. In the GA model, array data is partitioned into patches that are distributed across all nodes in the computation and are remotely accessible through one-sided operations.

B. Task Parallel Programming Model

The task parallel programming model provides a globally accessible, but potentially distributed, pool of available tasks that the programmer can add new tasks into and remove tasks from for execution. In general, the job of selecting which task should be extracted by a particular processor is performed by the runtime system. This system is permitted to move tasks around in order to satisfy multiple criteria including load balance and locality. A popular and highly scalable load balancing strategy used by many task pools is work stealing [1].

In this work, we restrict the full task pool model by separating the operations of populating the task pool with work and processing the tasks in the pool into distinct phases. In particular, we do not permit tasks to add new sub-tasks into the pool. This restriction allows us to focus on techniques for maintaining consistent data produced by tasks. In future work, we plan to investigate how these techniques for data consistency can also be applied to sub-tasks that are generated as task outputs.

We consider a task model where all input and output data is stored in global arrays. We require that tasks specify their inputs and outputs up front so that they can be managed by the fault tolerant runtime system. In addition, all input data must be read-only and all output data must be write-only. At present we focus on an output mode where tasks accumulate results into an output array using GA atomic accumulate operations. This model is sufficient for a broad class of iterative solvers and enables the system to track output completion of tasks in order to detect corruption and incompleteness due to failure.

C. Failure Model

In this work we focus on node failures and assume a pool of spare nodes that is available to the runtime system. These nodes can be used to replace failed nodes, keeping the total number of nodes in the computation fixed. In addition, we assume a fail-stop mode of failure where failed nodes crash and exit the computation. When a failure occurs, this results in the loss of a memory domain in global address space or, at the GA level, a patch of one or more global arrays.

D. Fault Tolerance Framework

Our fault tolerant task pool system assumes a fault tolerant runtime stack. It requires fault tolerance from Global Arrays and GA's communication layer, ARMCI, in order to support respawning failed nodes and re-establishing one-sided communication. In addition, a fault tolerant process management system is needed that can detect failures, respawn failed processes and inform the programming model that a failure has occurred. These systems are currently under development but not yet complete enough to evaluate our system. Thus, in this evaluation we rely on simulated faults as described in Section VI-B.

Fault tolerant task pools are also targeted at supporting fine-grain restart for sections of code that can be expressed using task pools. This system works in collaboration with existing coarse-grain checkpoint/restart techniques that are used to protect non- task parallel code sections. We use such a checkpoint/restart system to take a checkpoint prior to entering a task pool. When a failure occurs within the task parallel region, only the failed process' state is restored from checkpoint and it re-connects with the currently executing task pool.

III. Tracking Task Execution

When recovering from a failure, we wish to scan the set of tasks to determine which tasks have completed and which tasks are incomplete either because they did not execute fully or because some or all of the data they produced was lost. Tasks are permitted to produce outputs to regions of global arrays that span multiple memory domains. Thus, it is possible for a task that initially executed successfully to become partially or fully incomplete due to data loss from a failure.

In order to successfully detect this situation, we maintain a completion marker for every memory domain that a task writes to. This information makes it possible to detect partially complete tasks and reduce the number of cascading failures due to data corruption. Using the information on which patches of the global address space have been lost, tasks can be re-executed and re-write (i.e. atomic accumulate) outputs only to incomplete or recovered memory domains.

A. Distributed Task Execution Metadata

Given a set of tasks $T = \{t_1, ..., t_M\}$, a task T_i produces outputs to a set of memory domains, O_i. A task can produce at most N outputs, where N is the number of memory domains. Thus, we can organize metadata to track the completion of these outputs by forming $\mathbf{M}$, a $M \times N$ matrix where the tasks ids are the rows indices of the matrix and, for task T_i, $\mathbf{M}_{i,j}$ indicates the state of task i with respect to memory domain j.

Each $\mathbf{M}_{i,j}$ may contain one of the following values that indicates the state of task T_i with respect to memory domain j: $\perp$, *started*, *completed*. The initial value of all entries is $\mathbf{M}_{i,j} = \perp$, which indicates that task i has not produced an output to domain j. *Started* indicates that an update has been started and *completed* indicates that the previously *started* update has completed. During recovery, the system must ensure:

$$\forall T_i \in T, \forall j \in O_i \; \mathbf{M}_{i,j} = completed$$

$$\forall T_i \in T, \forall j \notin O_i \; \mathbf{M}_{i,j} = \perp$$

When a value of $\mathbf{M}_{i,j} = started$ is seen during recovery, this indicates that a failure occurred while task i was updating domain j and that domain j has become corrupted as a result of this failure. Thus, domain j must be restored from the checkpoint taken at the start of the task parallel region and row j of $\mathbf{M}$ must be reset to $\perp$ to avoid erroneous results. When $\mathbf{M}_{i,j} = \perp$ and $j \in O_i$, task j has not completed with respect to output j and should be scheduled to execution to produce this output.

B. Fault Resistant Metadata Storage

In order to avoid loss of important metadata when failure occurs, the metadata matrix $\mathbf{M}$ is distributed across memory domains where domain j stores the data in column j of $\mathbf{M}$. Thus, only the metadata corresponding to data stored on domain j is stored on domain j. In the event that domain j fails, all updates to j are lost and only the metadata corresponding to j's lost updates is lost. This co-location of metadata with the information that it tracks ensures that, at any given time, the state of all correct nodes is fully described by the available metadata.

IV. Fault Tolerant Task Execution

In Algorithm 1 we present the fault tolerant task execution algorithm. This algorithm ensures all tasks in a task pool execute successfully and that they produce a consistent result. We divide this algorithm into three parts, covered in detail in the following sections: (IV-A) task pool execution, (IV-B) data corruption detection, and (IV-C) completeness detection. At a high level, this algorithm proceeds by executing all unfinished tasks and checking to determine if a failure has occurred. If it has, it uses the metadata to detect data corruption and incomplete tasks. All incomplete tasks are then re-entered into the task pool and processing repeats until execution succeeds with no failures.

Algorithm 1 Fault tolerant task pool execution algorithm.

```
let: $me$ be this process' rank
let: $nproc$ be the total number of processes
$failed \leftarrow false$
$TP \leftarrow T$
$CP \leftarrow checkpoint()$
repeat
    $TP.process()$ // Execute all tasks in the task pool
    $failed \leftarrow detect\_failure()$
    if $failed = true$ then
        $TP.scan()$ // Detect corruption due to failures
        // Add all incomplete tasks back to task pool
        $TP.check\_completion()$
    end if
until $failed = false$
```

A. Task Pool Processing

Executing all of the tasks, or processing, a task pool requires performing a series of actions defined by each task object. In our task model, we require that all tasks in the task pool can be executed concurrently and prohibit tasks from producing sub-tasks. The algorithm for processing the task pool is given in Algorithm 2. This algorithm fetches the next available task from the task pool and executes it. When no more tasks are available, all processes line up at a barrier to wait for completion of all tasks before proceeding. If the underlying runtime system requires collective recovery, it can be triggered during this collective step.

We break down task execution into three steps: fetching task inputs, performing task execution, and writing task outputs. During execution, a task is not permitted to produce any outputs into the global address space. Instead, this is handled by the runtime system to allow for additional bookkeeping needed to detect data corruption if a failure occurs during the write operation. In our model, task outputs are specified up front by the user and metadata is used to determine which of these outputs need to be written (i.e. accumulated) to avoid producing an incorrect result. In order to avoid extra communication, this information is generally encoded into the task descriptor rather than gathering potentially distributed metadata in-line.

Algorithm 2 *TP.process*: Execute all tasks in a task pool

```
while $t_i \leftarrow TP.next()$ do
    $t.fetch\_inputs()$
    $t.execute()$
    for all $j \in O_i$ do
        if $\mathbf{M}_{i,j} = \perp$ then
            $\mathbf{M}_{i,j} \leftarrow started$
            $contribute(j)$
            $\mathbf{M}_{i,j} \leftarrow completed$
        end if
    end for
end while
$barrier()$
```

B. Corruption Detection

After the task pool has been processed we check to determine if a failure has occurred. If one has, we must check to determine if it occurred during a write operation resulting in an incomplete write and data corruption. This requires a scan of local metadata to detect any entries in the $started$ state, indicating that they were started but not completed. If such an entry exists, this node must perform recovery by restoring its shared data from the checkpoint taken at the start of the task parallel region and resetting the corresponding metadata entries. This scan is necessary because we allow read-modify-write (i.e. accumulate) updates of shared data. If only non-overlapping writes were allowed then this step would not be necessary.

Algorithm 3 *TP.scan*: Detect data corruption due to failure and initiate recovery.

```
for $i = 1 \ldots M$ do
    if $M_{i,me} = started$ then
        $recover()$ // Restore from $CP$ and $M_{*,me} \leftarrow \perp$
    end if
end for
```

C. Completeness Detection

When a failure has occurred, we perform analysis on the metadata to determine which tasks are incomplete due to the failure. Incomplete tasks may need to re-compute some or all of their outputs and are added back into the task pool for re-execution during the next round. We next describe two algorithms to perform this step: a simple naive algorithm and a home-based algorithm that greatly improves on the communication efficiency of the naive approach.

1) Naive Algorithm: In Algorithm 4 we present a simple approach to detecting incomplete tasks. In this approach, for every task we check the metadata entries corresponding to the elements of its output set. If one of these entries is not

$completed$ the task is re-entered into the task pool. This can be done in parallel for each task in the set of tasks, preserving the property that a task can be added at most once to the task pool.

Algorithm 4 *TP.check_completion* (naive): Find incomplete tasks and add to the task pool.

```
for all $T_i \in T$ do
    for $j \in O_i$ do
        if $\mathbf{M}_{i,j} \neq completed$ then
            $TP \leftarrow TP \cup T_i$
            break
        end if
    end for
end for
```

2) Home-Based Algorithm: The number of communication steps required for the naive approach is proportional to $O(|T| \cdot |O_{avg}|)$ where T is the set of tasks and O_{avg} is the average number of outputs per task. The size of T is generally much larger than the number of processors to allow for load balancing and continues to grow with problem size. In addition, due to data distribution of shared arrays, O_{avg} also grows with system size for a given problem. This gives a worst case communication complexity that is $O(nproc^2)$ steps. Thus, a more scalable solution is needed.

In Algorithm 5 we present an algorithm that introduces the notion of *recovery homes* in order to reduce the communication complexity of scanning the metadata. In this scheme, the set of tasks is partitioned according to task ids (row numbers in the metadata matrix) and each block of tasks is assigned to a process who is the home for these tasks. The assignment of tasks to processes is known by all processes and a hashing function H can be used to map task ids to their recovery homes. During recovery, each process scans its local metadata and compiles a message for each home indicating which tasks homed there have $completed$ metadata entries on the scanning node. These messages are then sent to all the homes and used by the home to check for task completion. Thus, each process sends one message to every other process resulting in a total of $O(nproc)$ communication steps.

Algorithm 5 *TP.check_completion* (home-based): Find incomplete tasks and add to the task pool.

```
let: $T' \subseteq T$ be the set of tasks homed on $me$
let: $H : \mathbb{N} \mapsto rank$
let: $MSG$ be an $M \times MAX\_OUTPUTS$ matrix
let: $M'$ be an $nproc \times MAX\_OUTPUTS$ matrix
for $i \in 1 \ldots M$ do
    if $\mathbf{M}_{i,me} = completed$ then
        Append $i$ to $MSG[H(i), *]$
    end if
end for
$M' \leftarrow$ All-to-All exchange of $MSG$
for $T_i \in T'$ do
    for $j \in O_i$ do
        if $j \cap M'[i] = \emptyset$ then
            $TP \leftarrow TP \cup T_i$
            break
        end if
    end for
end for
```

V. Discussion

The fault tolerant task pool model yields several attractive properties which we describe in this section.

A. Tolerance of an arbitrary number of failures

The metadata tracking scheme described in Section III is robust in the presence of an arbitrary number of failures. Because the bookkeeping information lost due to failure corresponds only to data that is also lost, at any given time all correct nodes contain metadata that describes the complete state of the valid data in the computation.

B. Low performance overhead when no failures occur

The task execution algorithm presented in Section IV adds only the overhead of maintaining the metadata to correct processes. This involves two sets of communication operations per task: one to set metadata bits to $started$ before writing outputs and one to set them to $completed$ when finished. These communication operations can be overlapped with task execution without any impact on robustness. Metadata bits can be set to $started$ using non-blocking communication at the start of task execution and setting them to $completed$ can be overlapped with execution of the next task. In this work we have focused on the cost of recovery and have not yet fully explored such techniques to mitigate the overhead of metadata maintenance. Thus, for the experiments where no faults occurred communication is still performed to maintain metadata. For SCF, this has only a small impact on performance because it's tasks are long running, however applications with different characteristics could require more aggressive overhead management techniques.

C. Space overhead proportional to task pool size

Conventional in-memory checkpointing techniques can also be used for selective restart, however these techniques store full copies of critical data and can double a program's space requirements. In comparison, fault tolerant task pools provide finer-grain recovery while potentially using only a fraction of the space. The storage overhead for fault tolerant task pools is proportional to the size of the metadata matrix: the number of tasks multiplied by the number of outputs per task.

D. Recovery cost proportional to degree of failure

Conventional checkpoint/restart techniques respond to failure by restarting all nodes in the computation from the most recent checkpoint. The cost of this recovery mode is proportional to the total system size. In comparison, fine-grain recovery using the task pool model involves recovering only the failed nodes and recomputing only the data lost from those nodes. In addition, the task pool model provides load balancing, allowing all processes help with recomputation further accelerating this process.

E. *Bounded cascading failure*

Recovery is initiated on non-failed nodes to correct data corruption due to a failure that occurred while another node was writing to its memory domain. Because this recovery is isolated and does not trigger recovery on other nodes, the number of cascading failures is bounded by the maximum number of memory domains a task is permitted to write to concurrently: the maximum number of task outputs.

VI. Experimental Evaluation

In this section we present an experimental evaluation of our fault tolerant task pool system using the Self Consistent Field (SCF) computation from the quantum chemistry domain. These experiments were performed on a 650 node IBM System 1350 cluster with Infiniband interconnect. Nodes in this system are configured with two quad-core 2.5 GHz AMD opteron processors and 24GB of RAM.

A. *Self Consistent Field Computation*

The Self Consistent Field (SCF) computation is a key, computationally intensive step in many computational chemistry applications that computes the ground-state wave function for a system of atoms. In particular, SCF forms the starting point for most *ab initio* quantum chemistry simulations. In order to evaluate our system, we have extended an existing Global Arrays implementation of the closed-shell SCF method [4] with support for fault tolerant task pools.

B. *Fault simulation*

A full fault-tolerant infrastructure for GA is currently under development. Thus, in order to perform an early evaluation of our fault tolerant task pool we have simulated failures in software. This is done at the user level by restoring a node's patch of the global array from a checkpoint and clearing the corresponding metadata. Worst case failures are simulated in order to elicit the upper bound on the cost of recovery. We define a worst case failure as a failure that occurs after the last task has finished executing. Thus, the maximum amount of data is lost by the failing process.

In Figure 1 we show the percent of all tasks that must be re-executed for SCF on a 48 beryllium atom data set experiencing a random worst-case single process failure. From this data we observe that the number of re-executed tasks is proportional to the number of processors via the data distribution. Due to the amount of global arrays data per process, a low processor count incurs a higher penalty due to failure than higher processor counts where the same global arrays are spread across more memory domains.

C. *Performance Study*

In Figures 2 and 3 we present the results from strong scaling experiments for the SCF computation on a system of 48 beryllium atoms on up to 256 processors. In Figure 2 we show the average execution time per iteration for the SCF kernel and in Figure 3 we show the relative performance as the ratio of the performance relative to the performance when no faults occur. Each graph shows data for three schemes: a baseline

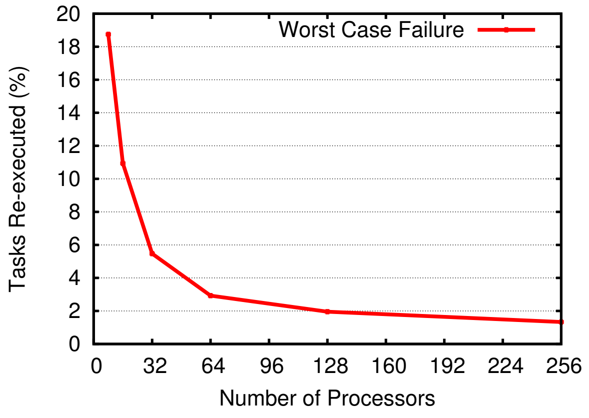

Fig. 1. Penalty in tasks re-executed for a worst-case single process failure on SCF.

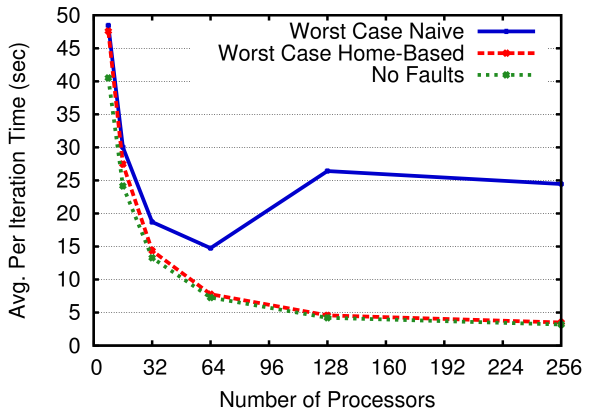

Fig. 2. Execution time for no faults and worst case failures using naive and home-based recovery schemes.

scheme where no failures occur, worst case failures with naive completeness detection, and worst case failures with the home-based completeness detection. For schemes that experience faults, we simulate one worst case failure per iteration of the SCF kernel.

By looking at the execution time for the naive completeness detection scheme, we see that the $O(nproc^2)$ cost for recovery can quickly dominate execution time on larger processor counts resulting in substantial slowdown. The home-based scheme offers the best performance in the presence of failures and closely follows the baseline performance trend. From the slowdown data, we see that the average performance penalty to recover from a worst-case single process failure is less than 10% for the home-based scheme while it grows very large for the naive scheme. In comparison, a conventional checkpoint/restart scheme incurs a penalty of 50% slowdown for a worst case failure because it must repeat the computation a second time.

VII. Related Work

User-directed checkpoint/restart techniques such as Berkeley Lab's Checkpoint/Restart [5] and GA checkpoint/restart [6] commit process checkpoints to disk periodically. Other

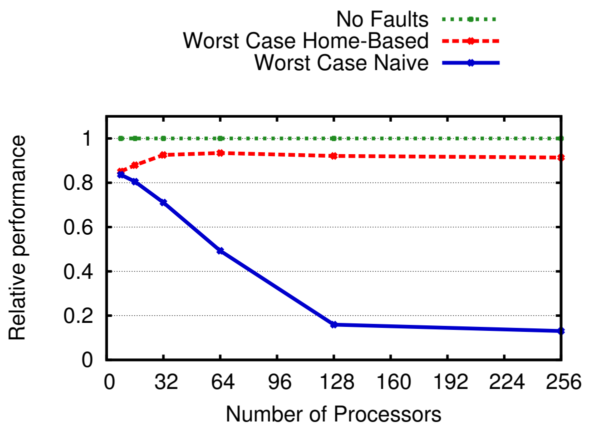

Fig. 3. SCF performance relative to the case with no faults.

process-level checkpoint/restart systems such as [7] and [8] perform checkpointing automatically throughout the computation but must log messages to ensure consistent global snapshots are captured. User-level checkpointing reduces the volume of data stored by storing only critical data and regenerating ancillary data during recovery. Some systems, notably NWChem [9], store a disk-resident database of intermediate results at each stage of the computation that can be used for restart. Fault tolerant task pools complement these coarse-grain recovery techniques with a fine-grain selective recovery mechanism that can be used to protect task parallel regions of a program against faults.

Efforts are also underway to develop a fault tolerant runtime stack, including a coordinated system-wide fault notification and handling backplane [10] as well as fault tolerant FT-MPI [11]. In addition to such reactive solutions, proactive fault tolerance [12] uses pre-failure feedback to prevent process failures via mechanisms like migration.

Cilk-NOW [13] provides fault tolerant task parallel computing on a network of workstations. However, Cilk-NOW is client-server based, focuses on functional parallelism, and does not support a shared global address space. In comparison, fault tolerant task pools has a fully distributed architecture and supports a shared global address space. BOINC [14] supports fault tolerant task parallel execution on a network of unreliable volunteer machines which can produce erroneous results due to failures. Thus, redundant computation must be performed in order to verify correct results for every unit of computation. BOINC is also client-server based and does not support a shared global address space.

Much work has been done to develop fault tolerance for the Linda programming model [15]. These efforts focus on maintaining a stable, content-addressable shared tuple space through replication and preventing tuple loss due to failure using atomic transactions. In contrast, this work uses the location-addressable global address space provided by GA and does not require transactions. Updates to shared data are performed via atomic accumulate, however failures can interrupt these operations resulting in memory corruption; metadata is used to identify and correct these inconsistencies. Linda also supports task parallelism through the shared tuple space. In contrast, FT task pools logically separates the tasks from the application's shared data and uses a checkpoint of the task list to detect lost or incomplete tasks, avoiding problems of duplication and task loss.

VIII. Conclusion

We have presented a new approach to fault tolerance that leverages the task pool programming model to survive an arbitrary number of failures. This system improves on conventional techniques by offering a cost for recovery that is proportional to the degree of failure while maintaining low overheads. Fine-grain recoverability is achieved by tracking individual task progress with distributed metadata. This system was evaluated on the SCF computational chemistry kernel and demonstrated to achieve recovery penalties of less than 10% for worst case failures on 256 processors.

References

[1] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan, "Scalable work stealing," in *Proc. 21st Intl. Conference on Supercomputing (SC)*, 2009.

[2] ——, "Scioto: A framework for global-view task parallelism," in *Proc. of 37th Intl. Conference on Parallel Processing (ICPP)*, 2008.

[3] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: a portable "shared-memory" programming model for distributed memory computers," in *Supercomputing (SC)*, 1994.

[4] J. L. Tilson, M. Minkoff, A. F. Wagner, R. Shepard, P. Sutton, R. J. Harrison, R. A. Kendall, and A. T. Wong, "High performance computational chemistry:(ii) a scalable SCF program," in *J. Computational Chemistry*, vol. 17, 1995, pp. 124–132.

[5] J. Duell, P. Hargrove, and E. Roman, "The design and implementation of Berkeley Lab's linux checkpoint/restart," Berkeley Lab, Tech. Rep. LBNL-49659, 2002.

[6] V. Tipparaju, M. Krishnan, B. Palmer, F. Petrini, and J. Nieplocha, "Towards fault resilient global arrays," in *Intl. Conf. on Parallel Computing (ParCo)*, 2007.

[7] Q. Gao, W. Yu, W. Huang, and D. K. Panda, "Application-transparent checkpoint/restart for MPI programs over infiniband," in *Proc. 35th Intl. Conf. on Parallel Processing (ICPP)*, 2006.

[8] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott, "Reliability-aware approach: An incremental checkpoint/restart model in hpc environments," in *Proc. 8th Intl. Symp. on Cluster Computing and the Grid (CCGRID)*, 2008.

[9] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, "High performance computational chemistry: An overview of NWChem a distributed parallel application," *Computer Physics Communications*, vol. 128, no. 1-2, pp. 260 – 283, 2000.

[10] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A coordinated infrastructure for fault-tolerant systems," in *Intl. Conf. on Parallel Processing (ICPP)*, 2009.

[11] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *Proc. 10th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[12] G. Vallee, K. Charoenpornwattana, C. Engelmann, A. Tikotekar, C. Leangsuksun, T. Naughton, and S. L. Scott, "A framework for proactive fault tolerance," in *Proc. 3rd Intl. Conf. on Availability, Reliability, and Security*, 2008.

[13] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *Proc. USENIX Annual Technical Conference*, 1997.

[14] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proc. 5th IEEE/ACM Intl. Workshop on Grid Computing (GRID)*. IEEE Computer Society, 2004, pp. 4–10.

[15] D. Bakken and R. Schlichting, "Supporting fault-tolerant parallel programming in Linda," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 3, pp. 287 –302, Mar 1995.